Support for Relaxed Currency and Consistency Constraints in MTCache

Hongfei Guo — University of Wisconsin — guo@cs.wisc.edu
Per-Åke Larson — Microsoft — palarson@microsoft.com
Raghu Ramakrishnan — University of Wisconsin — raghu@cs.wisc.edu
Jonathan Goldstein — Microsoft — jongold@microsoft.com

1. INTRODUCTION

A typical e-commerce site such as eBay makes frequent use of replicated and cached data. When browsing auctions in a category, the data (e.g., item prices, number of bids) may be a little out of date. However, most users understand and accept this, as long as the page they see when they click on an individual auction is completely current. As a concrete example, consider the following example query that returns a summary of books with the specified title:

```
Q1: SELECT     ...
    FROM       Books B, Reviews R
    WHERE      B.isbn = R.isbn and B.title = “Databases“
```

Different applications (Apps) may have different freshness requirements for this query. App A needs an up-to-date query result. App B prefers a quick response time but doesn’t care if the reviews are a bit old. App C does not mind if the result is stale but it requires the entire result to be snapshot consistent, i.e., reflect a state of the database at a certain point of time. App D is satisfied with a weaker version of this guarantee, requiring only that all rows retrieved for a given book reflect the same snapshot, with different books possibly from different snapshots.

Application designers normally understand when it is acceptable to use copies and what levels of data staleness and inconsistency are within the application’s requirements. Currently, such requirements are only implicitly expressed through the choice of data sources for queries. For example, if a query does not require completely up-to-date data, we may design the application to submit it to a database server C that stores replicated data instead of submitting it to database server B that maintains the up-to-date state. The routing decision is hardwired into the application and cannot be changed without changing the application.

This very much resembles the situation in the early days of database systems when programmers had to choose what indexes to use and how to join records. This was remedied by raising the level of abstraction, expressing queries in SQL and making the database system responsible for finding the best way to evaluate a query. We believe the time has come to raise the level of abstraction on the use of replicated and cached data by allowing applications to state their data currency and consistency (C&C) requirements explicitly and have the system take responsibility for producing results that meet the requirements.

We propose that applications make the requirements known to the DBMS through explicit C&C constraints in queries and have developed mechanisms to guarantee that the constraints are satisfied. This not only provides a solid semantic foundation for the use of replicated and cached data and increases the robustness of applications, but it also opens the door for the DBMS to do C&C-aware cache management and replica maintenance. This paper provides a brief overview of our proposal and implementation in MTCache; see [GLRG04] for more detail and a comparison with related work.

2. SPECIFYING C&C CONSTRAINTS

We first clarify what we mean by the terms currency and consistency. Suppose we have a database with two tables, Books and Reviews, as might be used by a small online book store, which are managed by a back-end database server.

Replicated data or the cached result of a query may not be completely up to date. Currency simply refers to how current or up-to-date a set of rows are (a table, a view or a query result). We define it as the elapsed time since a copy became stale.

Suppose that we have two replicated tables BooksCopy and ReviewsCopy. The state of each corresponds to some snapshot of the back-end database. If the states of the two replicas reflect the same snapshot, we say that they are mutually consistent or that they belong to the same consistency class.

To express C&C constraints we propose a new currency clause for SQL queries. We’ll use Q1 to illustrate and explain different forms of the currency clause, as shown in Figure 1.

```
E1: CURRENCY BOUND 10 min ON (B, R)
E2: CURRENCY BOUND 10 min ON (B), 30 min ON (R)
E3: CURRENCY BOUND 10 min ON (B) BY B.isbn,
                   30 min ON (R) BY R.isbn
E4: CURRENCY BOUND 10 min ON (B, R) BY B.isbn
```

Figure 1: Example C&C constraints

First consider currency clause E1. It expresses two constraints: a) inputs cannot be more than 10 min out of date and b) the states of the two input tables must be consistent. Enclosing two or more tables in parenthesis indicates that they are in the same consistency class and must be mutually consistent. To be correct, the result obtained using any replicas must be equivalent to the result obtained if the query were computed against snapshots of Books and Reviews taken from the same database state and no older than 10 min.

E2 relaxes the currency bound on R to 30 min and no longer requires that the tables be from the same snapshot by placing them in different consistency groups.

We assume that isbn is a unique key of Books. E3 allows each row of the Books table to originate from different snapshots. The phrase “R by R.isbn” has the following meaning: if the rows in Reviews are grouped on isbn, rows within the same group must originate from the same snapshot. Note that a Books row and the Review rows it joins with may be from different snapshots (because Books and Reviews are in different consistency classes). Compare this with E4, which requires that each Books row be consistent with the Reviews rows that it joins with. However, different Books rows may be from different snapshots.

In summary, a C&C constraint consists of a set of triples where each triple specifies (1) a currency bound; (2) a set of tables forming a consistency class; and (3) a set of columns defining how to group the rows of the consistency class into consistency groups. A SQL query may of course contain multiple SFW blocks. C&C constraints are not restricted to the outermost block of a query — any Select-From-Where block can have a C&C constraint. If a query contains multiple blocks with currency constraints, *all* constraints must be satisfied.

3. IMPLEMENTATION IN SQL SERVER

We have extended MTCache, our mid-tier database cache prototype described in [LGZ04], to support queries with C&C constraints. To this end, we keep track of which local materialized views are mutually consistent and how current their data is. We extended the cost-based optimizer to take into account the query’s C&C constraints and the C&C properties of applicable local materialized views during optimization. In contrast to traditional plans, plans must now include operators for performing runtime checking of the currency of local data. Depending on the outcome of this test, the plan switches between using either local data or remote data.

C&C Tracking Mechanism We group local materialized views into (logical) currency regions (CRs). The update mechanism used (transactional replication) propagates changes asynchronously but ensures that the views within a CR are transactionally and mutually consistent at all times. The data in a view may be somewhat out of date but it is always a transactionally correct snapshot of the underlying database.

We have a global heartbeat table at the back-end containing one row and one column: the current timestamp, which is updated at regular intervals by a stored procedure. This table is replicated into local heartbeat tables, one for each currency region. Each currency region is associated with a distribution agent that wakes up at regular intervals and propagates all pending changes, including changes to the heartbeat table. We can thus guarantee that a view is up to date as of the time found in its region’s heartbeat table because all changes up to that time have been propagated to the views.

Consistency Checking Mechanism We enforce consistency constraints at optimization time by making use of the optimizer’s plan property mechanisms. C&C constraints are captured during parsing, checked and normalized into a single constraint. The normalized C&C constraint is attached as a required C&C property to the root and inherited recursively by its children.

As part of delivered plan properties, C&C properties are computed bottom-up while building a physical plan. Each physical operator (select, hash join etc.) computes what properties it delivers given the properties of its inputs. Whenever a new root operator is added to the plan, we check whether the resulting plan satisfies all required C&C properties. If not, the new plan, i.e., the root operator, is discarded. Among the qualifying plans, the one with the estimated lowest cost is selected.

Currency Checking Mechanism Currency constraints must be enforced during query execution. The optimizer produces plans containing SwitchUnion operators that first check whether a local view is sufficiently up to date and switches between using the local view and retrieving the data from the back-end server. Similar two-faced plans are used by DBCache [BAK+03].

We modified the view matching mechanism in SQL Server to add currency checking for local views. With the original view matching mechanism, when a matched local view V is found for expression E, a substitute expression E(V) is built. If a currency guard is required for V, we create the substitute shown in Figure 2 instead. The local plan is normal substitute E(V) while the remote plan consists of a remote SQL query created from expression E. If the currency guard evaluates to true that local plan is executed, otherwise the remote plan is executed.

The currency guard for a local view that belongs to region R is an expression equivalent to the following SQL predicate:

```
EXIST (  SELECT 1 FROM Heartbeat_R
         WHERE TimeStamp > getdate()–B )
```

where Heartbeat_R is the local heartbeat table for region R and B is the applicable currency bound from the query.

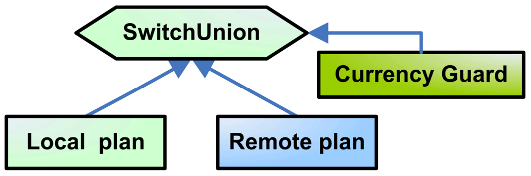

Figure 2: Substitute with SwitchUnion and a currency guard

Cost Estimation In most cases, using a local view is cheaper; but there are times when using remote sources on the back-end server turns out to be cheaper. For instance, there could be an applicable index on the back-end server but not locally. The optimizer selects the best plan (and subplans) based on cost so we need a way to estimate the cost of a SwitchUnion with a currency guard. We estimate the cost as

$$c = p * c_{local} + (1 - p) * c_{remote} + c_{cg}$$

where p is the probability of executing the local branch, c_{local} / c_{remote} is the cost of executing the local / remote branch, and c_{cg} the cost of checking the currency guard. The probability can be estimated as

$$\begin{array}{ll} p = 0 & \text{if } B\text{-}d \le 0 \\ p = (B\text{-}d)/h & \text{if } 0 \le B\text{-}d \le h \\ p = 1 & \text{if } B\text{-}d \ge h \end{array}$$

where B is the currency bound, d is the average propagation delay and h is the average propagation interval.

4. REFERENCES

[BAK+03] C. Bornhövd, M. Altinel, S. Krishnamurthy, C.Mohan, H. Pirahesh, and B. Reinwald. DBCache: Middle-Tier Database Caching For Highly Scalable E-Business Architectures. In *SIGMOD*, 2003.

[GLRG04] H. Guo, P. Larson, R. Ramakrishnan and J. Goldstein. Relaxed Currency and Consistency: How to Say “Good Enough” in SQL. In SIGMOD, 2004.

[LGZ04] P. Larson, J. Goldstein, and J. Zhou. MTCache: Transparent Mid-Tier Database Caching In Sql Server. In ICDE, 2004.